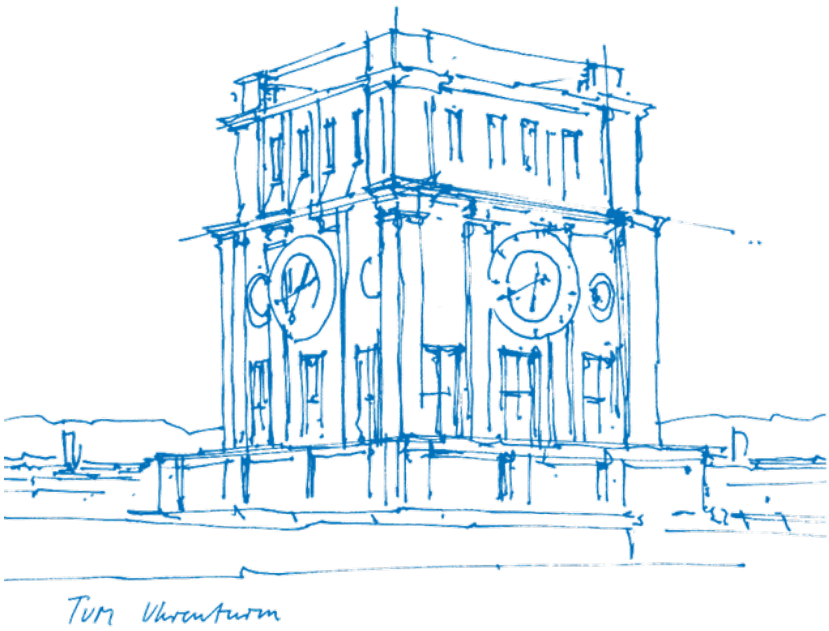
Chair of Geoinformatics
TUM Department of Aerospace and Geodesy
Technical University of Munich

Geodata for Testing Automated Driving Systems

Applied Geoinformatics 2

Czychon Lukas, Götzer Stephan and Rößl Florian

Degree program:	Geodesy and Geoinformatics
Supervisor:	M.Sc. Schwab, Benedikt Dr.-Ing. Donaubauer, Andreas
Submission:	Munich, 28.02.2021



Acknowledgments

Firstly we would like to thank Thomas Sedlmayer, a student working at PMSF, who gave us detailed information and access to PMSF OSI3 viewer. Furthermore our special gratitude goes to Benedikt Schwab for supervising this project. He gave us help and valuable input in weekly meetings to keep the project moving.

Contents

1 Introduction

In the module *Applied Geoinformatics 2* of winter semester 2020/21 students have to organize themselves into groups of three and select a project related to the field of Geographic Information System (GIS). The general tasks of the project can be described as follows:

- Development and prototypical implementation of a concept for solving the GIS related project.
- Documentation of the project.
- Presentation and discussion of scientific problems related to the project.

The following report should serve as documentation of the project "*Geodata for Testing Automated Driving Systems*".

Chapter 2 introduces the project with a motivation as well as a definition of the actual research questions. The standards studied, the tools used and the data available are then presented in chapter 3. Based on the presented standards follows the interoperability analysis in chapter 4, investigating differences and similarities of individual objects and their attributes in detail. In chapter 5 the design and concept of the implementation are represented. Afterwards chapter 6 presents results in form of validation and visualizations. Finally, the report is concluded with a summarizing conclusion in chapter 7.

2 Project Description

The development of autonomous cars brought up new challenges, on how to test new thoughts and algorithms safely. In order to be deemed reliable a vehicle has to drive several billion miles (Kalra, 2016). To avoid large sums of the budget being spent on in-situ testing, some parts can be validated in a simulation on a computer. Scenarios for such purposes have to be purposefully built and require a lot of attention to reassemble a detailed cityscape. It would therefore be useful to have a digital twin of a city that could be easily recreated from existing data.
This project tries to create such a digital twin. Therefore, it firstly analyses the provided data and then conceptualizes and implements a python package converting OpenDRIVE enhanced CityGML files through the use of 3DCityDB into a Open Simulation Interface (OSI) compliant format. The main tasks of the project are:

1. Analysis of provided CityGML Data (FME Data Inspector for data exploration)
2. Interoperability analysis between CityGML and OSI
3. Setup of a testing and development environment
4. Developing a CityGML to OSI converter/serializer
5. Visualizing the created OSI files in the ASAM OSI-Visualizer and the PMSF OSI-Viewer

3 Fundamentals

3.1 Standards

The following section introduces the involved standards, namely OpenDRIVE, CityGML and OpenSimulationInterface.

3.1.1 CityGML

"CityGML is an open data model and XML-based format for the storage and exchange of virtual 3D city models. It is an application schema for the Geography Markup Language (GML) [...] The aim of the development of CityGML is to reach a common definition of the basic entities, attributes, and relations of a 3D city model. This is especially important with respect to the cost-effective sustainable maintenance of 3D city models, allowing the reuse of the same data in different application fields" (*CityGML 2.0: OGC City Geography Markup Language (CityGML) Encoding Standard* n.d.).
The current version of CityGML is version 3.0 and can be found at `https://github.com/opengeospatial/CityGML-3.0CM`. It allows representing 3D geometry, 3D topology, semantics and appearance in different Level of Detail (LOD).

3.1.2 OpenX

Association for Standardization of Automation and Measuring Systems (ASAM e.V.) is a non-profit organization developing standards at implementation level in the field of automotive industry. The members of ASAM e.V. are international car manufacturers, suppliers and research institutes. The development of the standards is based on contributions of the members and open source community. Figure 3.1 visualizes a testing workflow and the role of the individual standards. Special attention should hereby be paid on the formats OpenDRIVE and OSI, as they are investigated in the upcoming chapters.

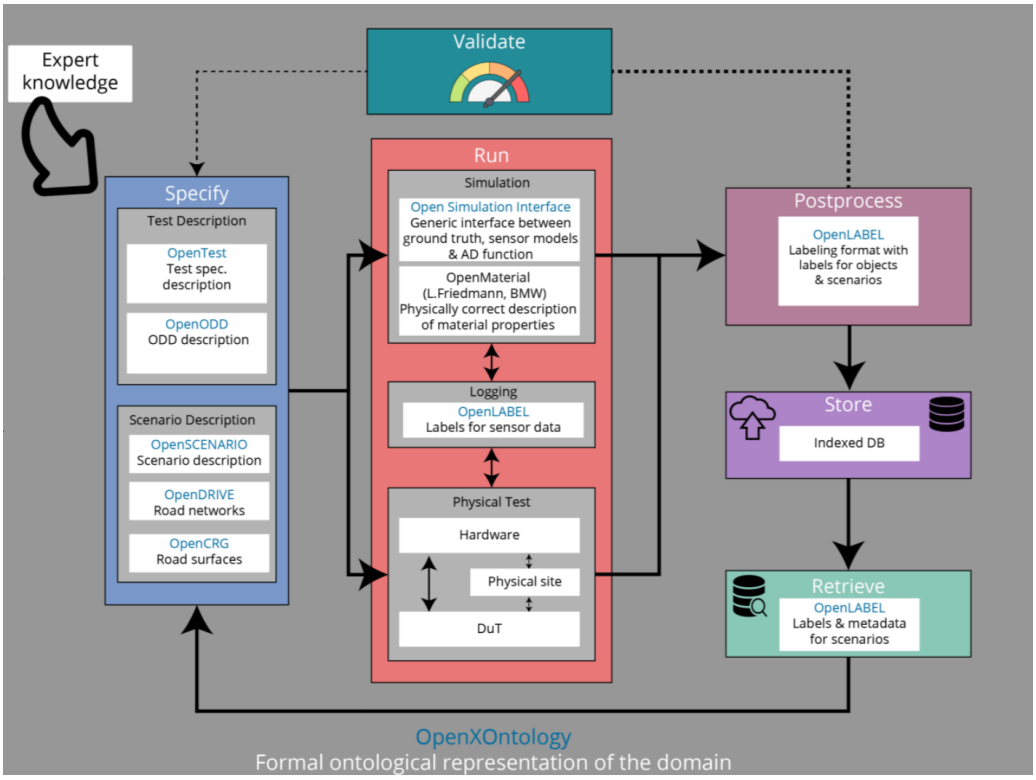


Figure 3.1 OpenX activities in a scenario-based testing workflow (ASAM e.V., 2020)

OpenDRIVE

"The OpenDRIVE format provides a common base for describing road networks with Extensible Markup Language (XML) syntax, with the file extension xodr. The data that is stored in an OpenDRIVE file describes the geometry of roads as well as features along the roads that influence the logics, for example, lanes and signals. The road networks that are described in the OpenDRIVE file can either be synthetic or real. The main purpose of OpenDRIVE is to provide a road network description that can be fed into simulations and to make these road network descriptions exchangeable." (ASAM e.V., 2020)

In general OpenDRIVE allows to define a static road network with basic elements, like Roads, Junctions or Controller. But it cannot model elements, which might interact with the road network (ASAM e.V., 2020).

OpenSimulationInterface

Open Simulation Interface (OSI) provides a generic interface between the function development framework and the simulation environment, in order to increase compatibility of automated driving functions and various driving simulation framework (Hanke et al., 2017).
Originally OSI was developed by Bayerische Motoren Werke (BMW) and Technical University Munich (TUM), but is now part of the ASAM Opendrive-X standards as this allows free public access and open source contribution, whereby the ASAM e.V. group will review contributions. The source code of the project and the corresponding documentation can be found at `https://github.com/OpenSimulationInterface`. The latest version is `v3.2.0 - OSI "Editorial Eaton"`, but a new version with a major release `V4.0.0` is already planned for July 2021.
The OSI standard defines two top level messages, namely `GroundTruth` and `SensorData` interface. The `GroundTruth` message enables the modeling of the real virtual environment of the simulation, which might be sent directly to a driving simulation framework or an additional consumer, who applies a sensor model with limited perception and disturbances to generate `SensorData` (Hanke et al., 2017). This workflow can also be seen in Figure 3.2.

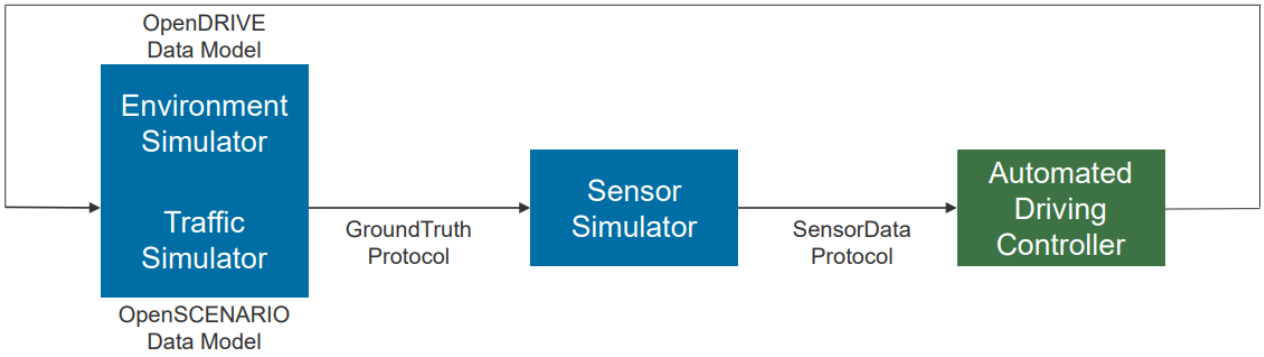


Figure 3.2 Exchanging data from the data model via OSI to a simulator (Thomsen, 2018)

OSI messages are based on protocol buffer version 2, which are developed and maintained by Google. "These protocol buffers are a language-neutral, platform neutral, extensible mechanism for serializing structured data - think XML, but smaller, faster and simpler" (Google Developers, 2008). At the beginning of year 2021 protocol buffers version 3 support the programming languages Java, Python, Objective-C, C++, Dart, Go, Ruby and C#.

3.2 Tools

The following tools, described in detail are used to store, handle, view and examine the Provided Data, detailed in the following section 3.3.

3.2.1 FME Data Inspector

Figure 3.3 shows the user interface of FME Data Inspector, an application of the FME Desktop software bundle, that is mostly for FME Workbench. The application is created by Safe Software Inc. that mainly focuses on the development of tools for feature manipulation.
FME Data Inspector is used in the project to get a first impression on the Provided Data. It also allows investigating how the OpenDRIVE information is integrated into a CityGML file. The last major use case in this project is the visual examination of the topology of different spatial objects.

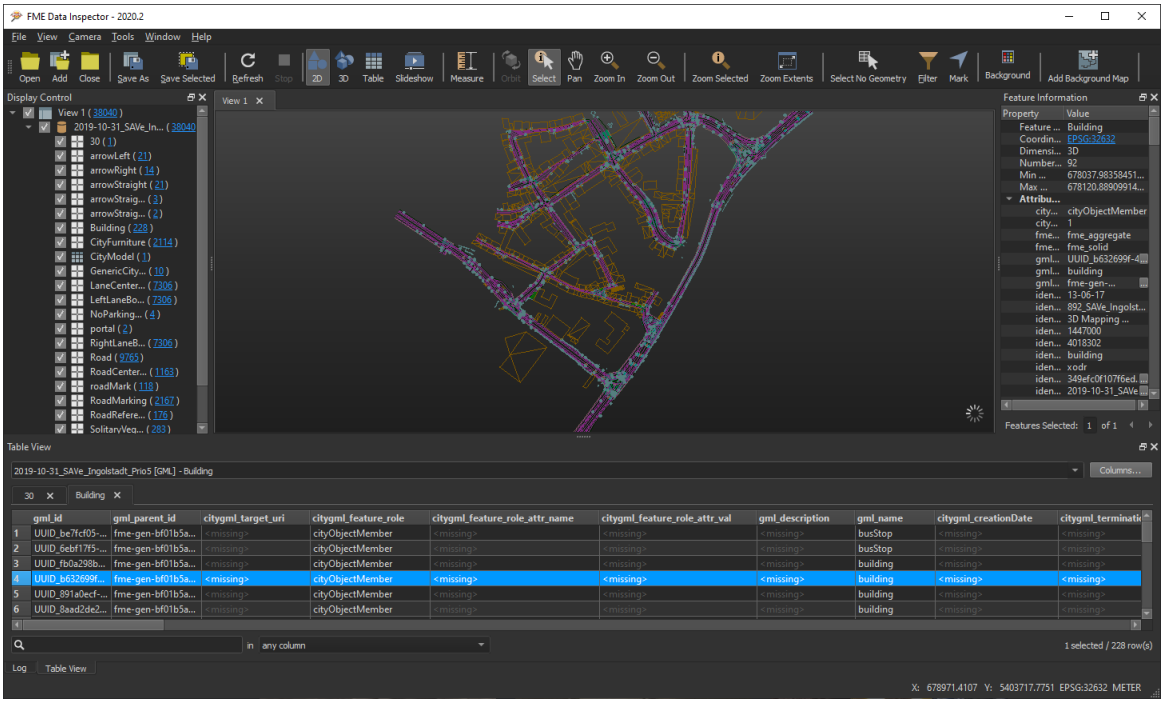


Figure 3.3 Screenshot of the FME Data Inspector user interface

3.2.2 3D City Database

"The 3D City Database (3DCityDB) is a free Open Source package consisting of a database schema and a set of software tools to import, manage, analyse, visualize, and export virtual 3D city models according to the CityGML standard. The database schema results from a mapping of the object ori- ented data model of CityGML 2.0 to the relational structure of a spatially-enhanced relational database management system (SRDBMS). The 3DCityDB supports the commercial SRDBMS Oracle (with Spatial or Locator license options) and the Open Source SRDBMS PostGIS (which is an extension to the free RDBMS PostgreSQL)." (Yao et al., 2021)

3DCityDB is the main application for data storage in the project, its purpose and use is later described in detail in chapter 5. The Provided Data is imported in to this database with the additional tool that is as well part of the 3DCityDB software package. Figure 3.4 shows the interface of the *3DCityDatabase Importer/Exporter*, a java based frontend that validates and imports CityGML files into the database.

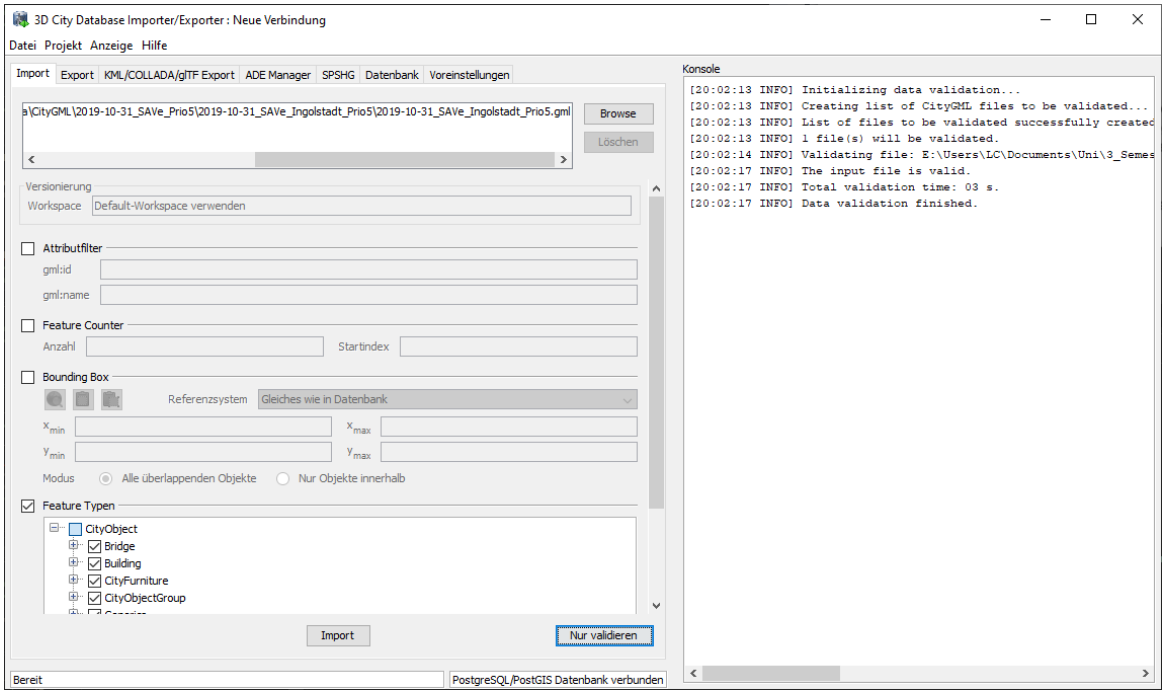


Figure 3.4 Screenshot of Importer user interface

3.2.3 PGAdmin

pgAdmin is an Open Source administration and development platform for PostgreSQL databases. It allows the administration of the database through a website hosted by the application. This tool is developed by a community and "is written in Python and jQuery with Bootstrap, using the Flask framework" (pgAdmin Development Team, 2021).
In this project pgAdmin is used to create materialized views and to investigate the data that is imported into the 3DCityDB. The most used feature can be seen in Figure 3.5, it allows the user to explore and debug spatial queries, by displaying the results on top of a Open Street Map (OSM) map layer.

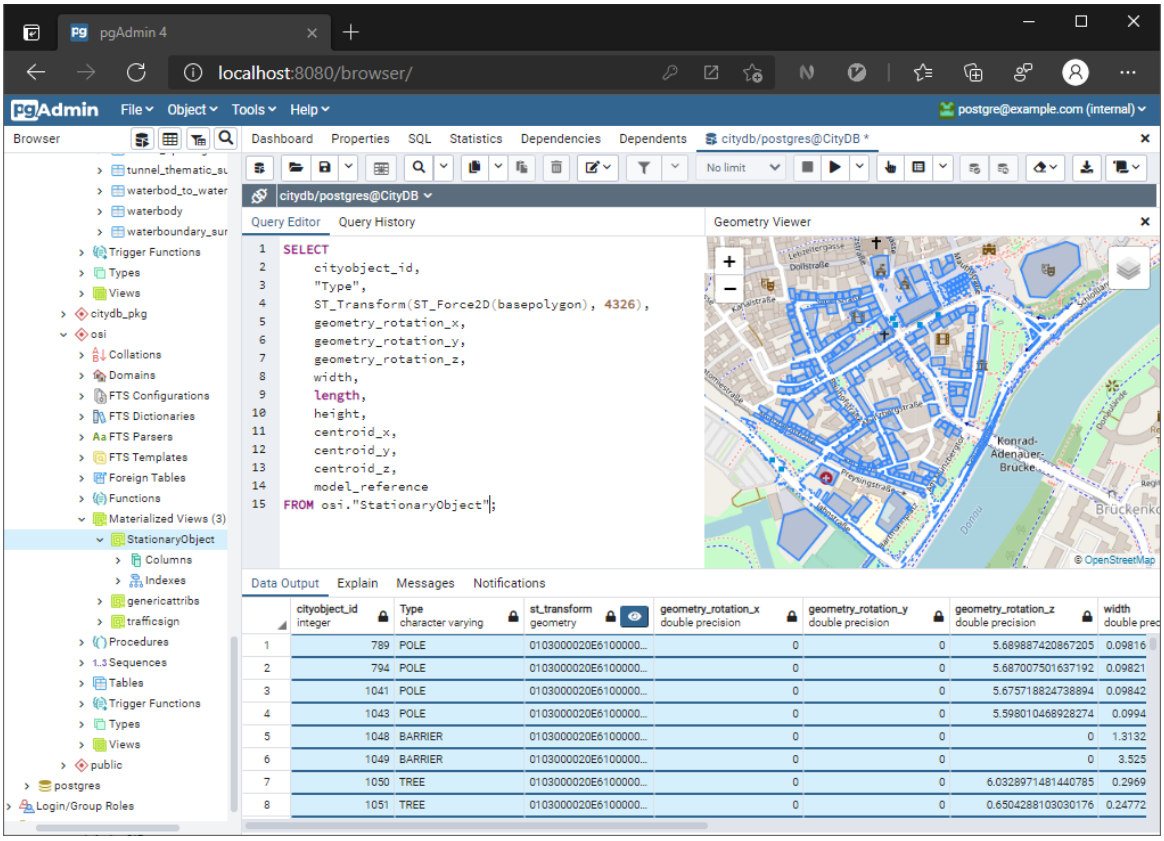


Figure 3.5 User interface of pgAdmin

3.2.4 Docker & Docker-Compose

Docker is a virtualization tool that is more lightweight than traditional virtual machines. It packages up software dependencies into a unit that allows them to run reliably on different computation environments. This unit gets defined as an image and later when it is executed in the Docker Engine it becomes a container. (Docker Inc., 2021)
Is a simple orchestration of containers defined in a `docker-compose.yml` file. Here multiple containers, known as services are composed together. This services can then be setup to work together, e.g. share the same network or the same environment variables. This can be used to define a system that delivers the same results even on different computers and therefore allows each group member to participate equally. In this project, Docker is primarily used to enable each project member to run the same consistent environment on its own computer. Another advantage on this approach is that the used tools do not collide with other installed tools. Below, Table 3.1 shows the defined services for the development environment in the afore mentioned `docker-compose.yml` file.

Service Name	Description	Base Image
3DCityDB	Geo database to store, represent, and manage virtual 3D city models.	tumgis/3dcitydb-postgis
PGAdmin	Website to access the underlying PostgreSQL database of the 3DCityDB.	dpage/pgadmin4
citygml2osi	Prototype written in Python that converts from CityGML to OSI.	python:3.9

Table 3.1 Used docker containers as stated in the `docker-compose.yml` file

3.3 Provided Data

The provided data as seen in Figure 3.6 is further used to analyse the interoperability of the OpenDRIVE enhanced CityGML format into the OSI format. The dataset describes the city centre of Ingolstadt in Bavaria and was produced by the conversion of a OpenDRIVE dataset with the r:trån tool developed by Schwab et al., 2020.

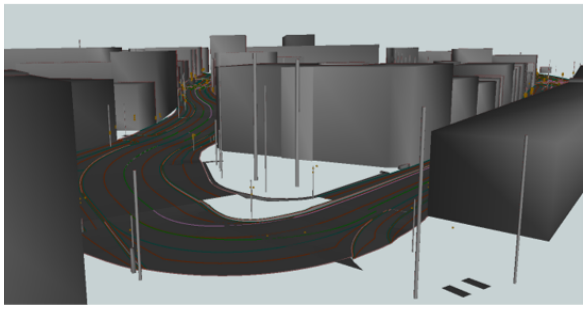


Figure 3.6 Screenshot of the provided data in the FME Data Inspector

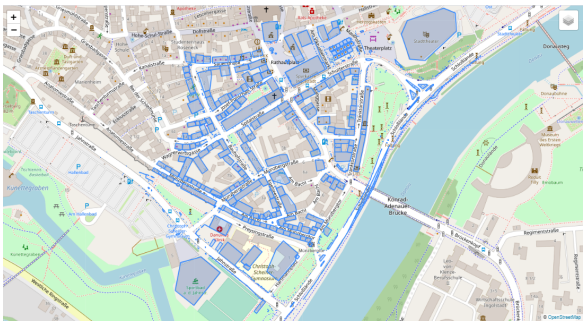


Figure 3.7 Screenshot of the provided data in PGAdmin

4 Interoperability Analysis

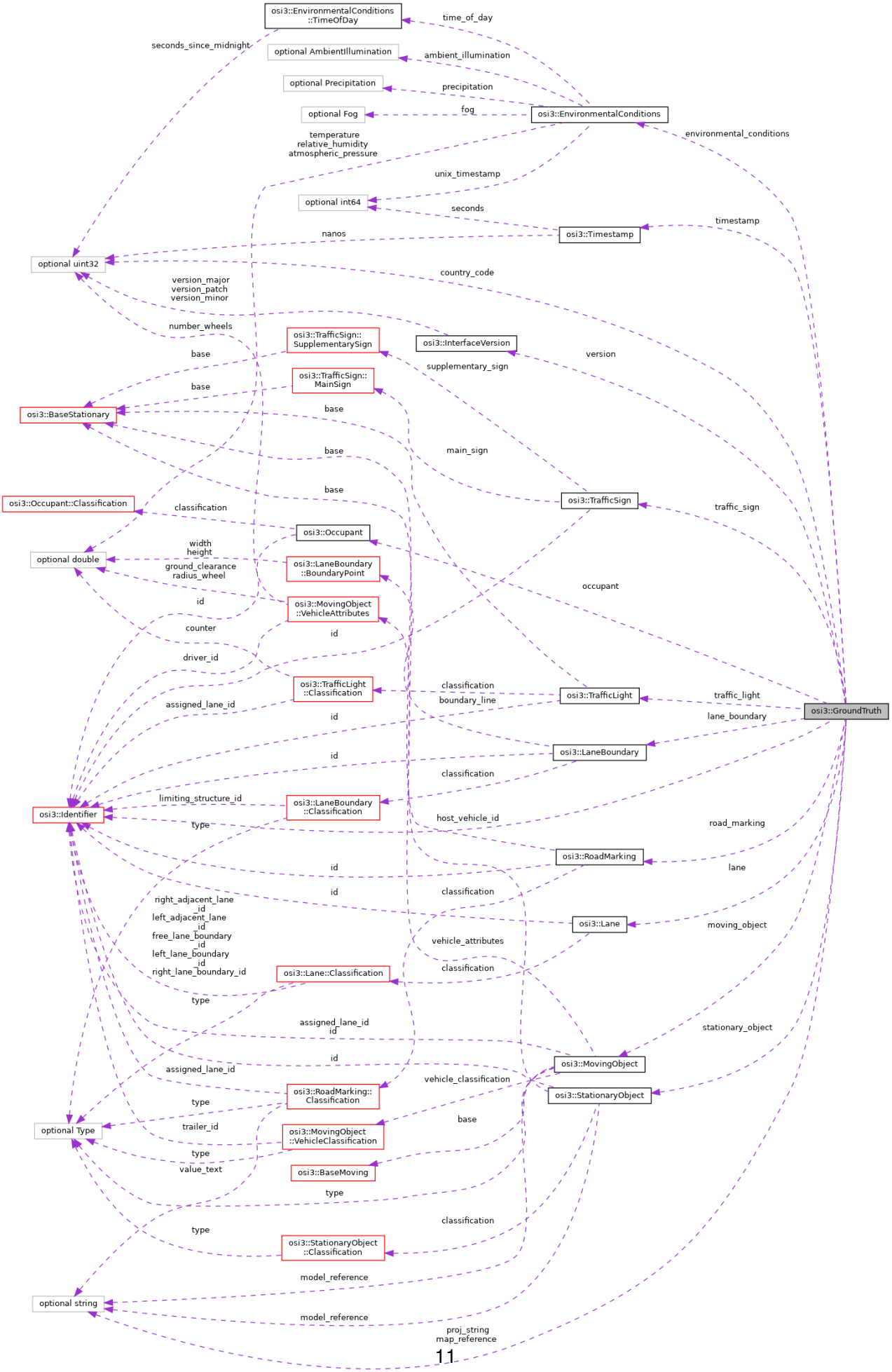
In order to check, whether it is feasible to convert CityGML data enriched by OpenDRIVE data into the OSI format, a so-called Interoperability analysis is conducted. Hereby a selected choice of the most relevant and promising OSI classes and their corresponding attributes of the `Ground Truth` message are investigated, if the required information is available in the CityGML dataset.

Table 4.1 presents the different indices describing the feasibility to convert CityGML to OSI objects and their corresponding attributes.

Abbr.	Description
DA	directly available
BF	built-in function can be used, e.g. PostGIS
CM	custom mapping is required
NA	not available (could be added to the dataset, e.g. color)
NP	not possible (dynamic attributes, e.g. temperature)

Table 4.1 Classification of the difficulty to map

Figure 4.1 visualizes the collaboration diagram of the `Ground Truth`. It is visible, that there are multiple fundamental types like `Identifier`, `Base Stationary` or `Timestamp` and base types like `string`, `double`, `integers` and predefined `Types` (enumerations), which together build more complex OSI classes.


osi3::EnvironmentalConditions ::TimeOfDay
time_of_day
seconds_since_midnight
optional AmbientIllumination
ambient_illumination
optional Precipitation
precipitation
optional Fog
fog
osi3::EnvironmentalConditions
environmental_conditions
temperature
relative_humidity
atmospheric_pressure
unix_timestamp
optional int64
seconds
osi3::Timestamp
timestamp
nanos
optional uint32
country_code
version_major
version_patch
version_minor
number_wheels
osi3::TrafficSign:: SupplementarySign
osi3::InterfaceVersion
version
base
supplementary_sign
osi3::TrafficSign:: MainSign
base
osi3::BaseStationary
base
base
main_sign
base
osi3::Occupant::Classification
classification
osi3::Occupant
osi3::TrafficSign
traffic_sign
width
height
ground_clearance
radius_wheel
optional double
osi3::LaneBoundary ::BoundaryPoint
occupant
id
osi3::MovingObject ::VehicleAttributes
counter
id
driver_id
osi3::TrafficLight ::Classification
classification
boundary_line
osi3::TrafficLight
traffic_light
osi3::GroundTruth
assigned_lane_id
id
lane_boundary
id
osi3::LaneBoundary
classification
limiting_structure_id
osi3::LaneBoundary ::Classification
osi3::Identifier
type
host_vehicle_id
road_marking
osi3::RoadMarking
id
lane
id
classification
right_adjacent_lane _id
left_adjacent_lane _id
free_lane_boundary _id
left_lane_boundary _id
right_lane_boundary_id
osi3::Lane
moving_object
vehicle_attributes
osi3::Lane::Classification
classification
type
assigned_lane_id
id
stationary_object
id
osi3::MovingObject
assigned_lane_id
vehicle_classification
osi3::RoadMarking:: Classification
osi3::StationaryObject
type
optional Type
trailer_id
base
osi3::MovingObject ::VehicleClassification
type
value_text
osi3::BaseMoving
type
classification
type
osi3::StationaryObject ::Classification
model_reference
optional string
model_reference
proj_string
map_reference

4.1 Stationary Object

OSI defines a `StationaryObject` as a simulated object, which is neither able to move nor it is traffic related object like `TrafficLight`, `TrafficSign` or `RoadMarking`. Figure 4.2 sketches the collaboration of the OSI `StationaryObject`, which is by Table 4.3 in detail analysed.

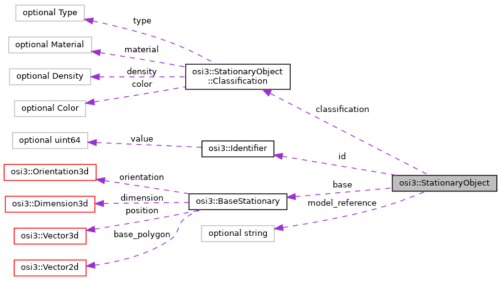


Figure 4.2 Collaboration diagram of the `StationaryObject`

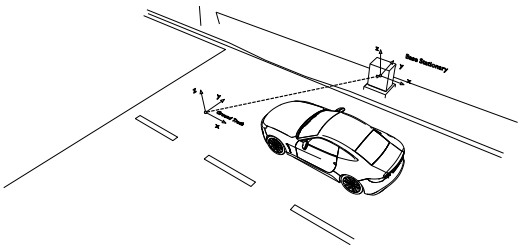


Figure 4.3 Illustration of the `StationaryObject`

StationaryObject	OSI Type	CityGML	IntIdx
id	`Identifier [uint64]`	`identifier_road_ObjectID`	DA
model_reference	`string`	`identifier_modelName`	DA
classification	`Classification`		
type	`Type [enum]`	`opendrive_roadObject_type`	CM[1]
material	`Material [enum]`	--	NA[2]
density	`Density [enum]`	--	NA
color	`Color [enum]`	--	NA
base	`BaseStationary`		
orientation	`Orientation3d`	`geometry_rotation`	DA
dimension	`Dimension3d`	`3D extent`	BF[3]
position (3D)	`Vector3d`	`center`	BF[4]
base_polygon	`rep. Vector2d`	`Surface_Geometry`	BF[4]

IntIdx Interoperability Index (IntIdx): abbreviations can be found in Table 4.1
bold text denotes a nested type
rep. denotes a repeated type
1 limited availabilty, as OSI supports only a limited number
2 might be assumed
3 via PostGIS: BOX3D
4 via PostGIS: Centroid
5 via PostGIS: ConvexHull

Table 4.3 StationaryObject interoperability table

In general the conversion from CityGML to OSI is straight forward, except the modelling of the geometry, in especially the calculation of the 2d contour, also known as `base_polygon`, turns out to be a problem as the polyhedral surface is not supported by every PostGIS function. Therefore, a workaround - `ST_FORCE2D(ST_CONVEXHULL(ST_COLLECT(surface_geometry.geometry))) AS basepolygon` - with the `surface geometry` is applied.

4.2 Lane

In the OSI standard the road network, in especially a road is described by individual `lanes`. In contrast to `OpenDrive`, which models junctions and lanes in different objects, these two objects are repre-

sented in the OSI standard by the `Lane` class with a different `Lane:Classification:Type`. A `lane`, characterized by its center line, knows about adjacent as well as antecessor and successor lanes. Figure 4.4 shows multiple lanes, for instance `l4`, which is defined by its dashed center line through the points $(cl4\_1, cl4\_2, cl4\_3, cl4\_4, cl4\_5)$.

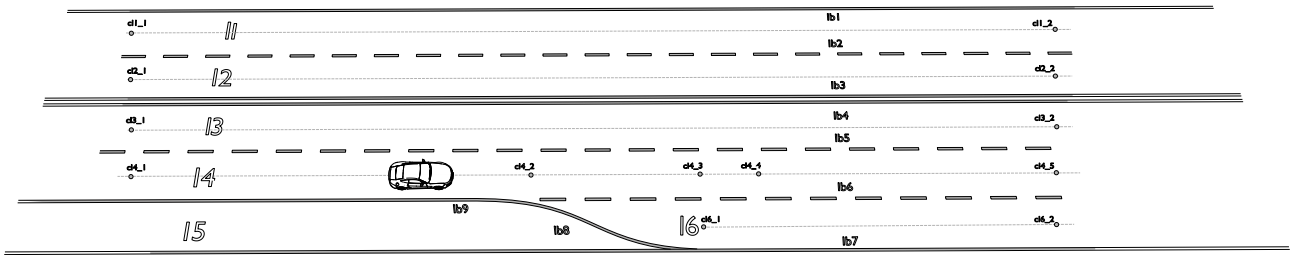


Figure 4.4 Illustration of the OSI `Lane`

Table 4.5 investigates the Interoperability of the OSI `Lane` and the `CityGML` data-set.

Lane	OSI	CityGML	IntIdx
id	`Identifier [uint64]`	`cityobject_id`	DA
classification	`Classification`		
type	`Type [enum]`	`map table name`	CM[1]
right_adjacent_lane_id	`rep. Identifier`	`cityobject_id`	CM[2]
left_adjacent_lane_id	`rep. Identifier`	`cityobject_id`	CM[2]
free_lane_boundary_id	`rep. Identifier`	`--`	NA
left_lane_boundary_id	`rep. Identifier`	`cityobject_id`	CM[3]
right_lane_boundary_id	`rep. Identifier`	`cityobject_id`	CM[3]
centerline	`rep. Vector3d`		BF
centerline_is_driving_direction	`bool`	`--`	NA
is_host_vehicle_lane	`bool`	`--`	NA
road_condition	`RoadCondition`		
surface_texture	`double`	`opendrive_lane_material_surface`	CM
surface_temperature	`double`	`--`	NP
surface_ice	`double`	`--`	NP
surface_roughness	`double`	`--`	NP
surface_freezing_point	`double`	`--`	NP
surface_water_film	`double`	`--`	NP
lane_pairing	`rep. LanePairing`		
successor_lane_id	`Identifier`	`cityobject_id`	CM[4]
antecessor_lane_id	`Identifier`	`cityobject_id`	CM[4]

IntIdx Interoperability Index (IntIdx): abbreviations can be found in Table 4.1
bold text denotes a nested type
rep. denotes a repeated type
1 Type depends on the table (Road or Junction)
2 Search for same `RoadId` and `LaneSectionId` but changed `LaneId` (± 1)
3 Table `left/right_lane_boundary` + same (`RoadId` + `LaneId` + `LaneSectionId`)
4 Change `LaneSectionId` (± 1) or check corresponding `openDrive_road_successor/predecessor_junction`

Table 4.5 Lane interoperability table

In order to model all the required relationships between different `lanes` and `lane_boundaries` a custom mapping after querying the database is necessary.

The city_objectID is generated while loading the CityGML dataset into the database. Another unique key candidate is the combination of table_name, identifier_RoadId, identifier_LaneId and the identifier_LaneSectionId, which also encodes a specific lane segment. Furthermore, is it necessary to consider the OpenDRIVE class Junction as OSI models junctions as lanes.

4.3 Lane Boundary

A OSI Lane Boundary determines the border and such the width of a OSI Lane. It is defined by its individual Boundary Points. Figure 4.5 displays a scene with multiple Lane Boundaries, like lb5, which is shared boundary of lane l3 and l4.

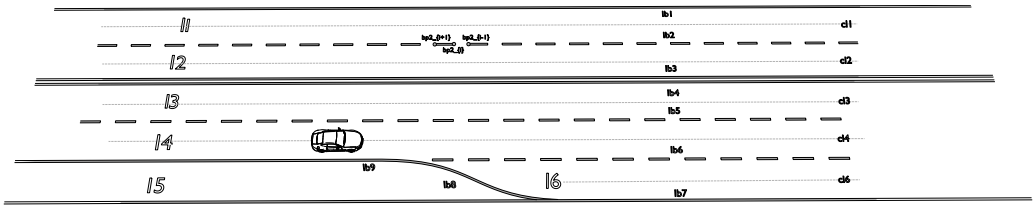


Figure 4.5 Lane Boundary

The interoperability analysis of Lane Boundary is shown by Table 4.7.

LaneBoundary	OSI	CityGML	IntIdx
id	Identifier [uint64]	cityobject_id	DA
classification	Classification		
type	Type [enum]	Road_Marking_Type	CM
color	Color [enum]	Road_Marking_Color	CM
limiting_structure_id	rep. Identifier	no topology available	NP
boundary_line	rep. BoundaryPoint[struct]		
width	double	-	NA
height	double	-	NA
position	Vector3d	LOD2_other_geom	BF

IntIdx Interoperability Index (IntIdx): abbreviations can be found in Table 4.1
bold text denotes a nested type
rep. denotes a repeated type

Table 4.7 Lane Boundary Interoperability table

The attribute Limiting Structure id could not be modelled as no information on a structure limiting the corresponding Lane is available.

4.4 Road Marking

The OSI RoadMarking class describes all lane surface markings, excluding lane markings which are described by the LaneBoundary class. RoadMarking also include text and speed limits that are written on the road.

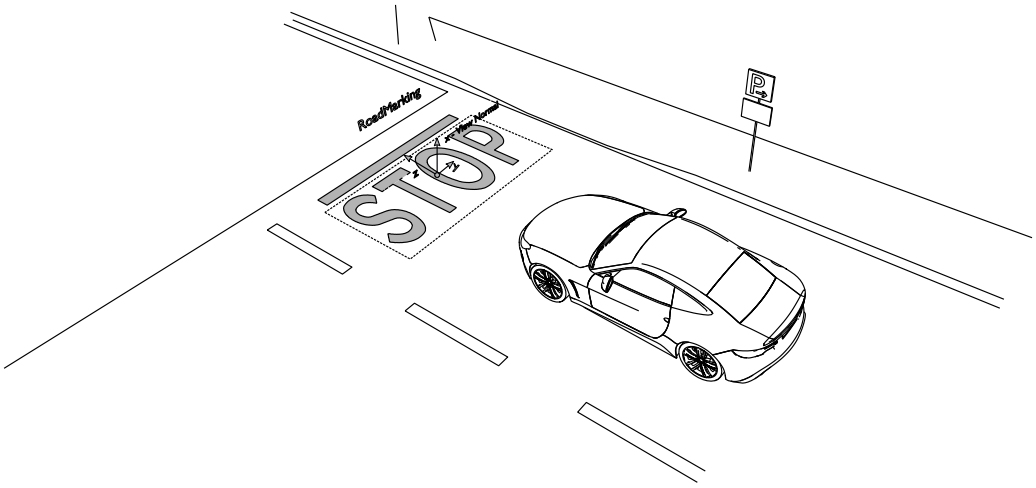


Figure 4.6 Road marking

RoadMarking	OSI	CityGML	IntIdx
id	Identifier [unit64]	identifier_roadObjectId	DA
classification	Classification[struct]		
type	Type[enum]	identifier_roadObjectName	CM
is_out_of_service	bool	-	NA
traffic_main_sign_type	Type[enum]	-	NA
value_text	string	-	NA
monochrome_color	Color[enum]	opendrive_roadMarking_color	CM
assigned_lane_id	rep. Identifier[struct]	identifier_laneId	DA
value	rep. TrafficSignValue[struct]		
text	string	-	NA
value_unit	Unit[enum]	-	NA
value	double	-	NA
base	BaseStationary[struct]		
orientation	Orientation3d[struct]	geometry_rotation_z, geometry_rotation_y, geometry_rotation_x	DA
dimension	Dimension3d[struct]	-	NA
position	Vector3d[struct]	lod2Geometry	CM
base_polygon	rep. Vector2d[struct]	lod2Geometry	DA

IntIdx Interoperability Index (IntIdx): abbreviations can be found in Table 4.1
bold text denotes a nested type
rep. denotes a repeated type

Table 4.9 Road Marking interoperability table

Some remarks:

- CityGML models road markings as Generic City Objects. There is no corresponding class to the OSI `RoadMarking` class in CityGML.
- As there is no class for road markings in CityGML the included objects, that can be classified as OSI `RoadMarking`, have no consistent mapping.
- The CityGML concept of road markings is fundamentally different from the concept in OSI

4.5 Traffic Sign

The OSI `TrafficSign` represents all kind of traffic signs found on the road. It differentiates between `MainSign` and `SupplementarySign`. The object describes the sign itself without a pole. The main sign can be modified by a supplementary sign. As an example, the main sign is a speed limit it can be enhanced by a supplementary sign, that adds a time limit to the speed limit.

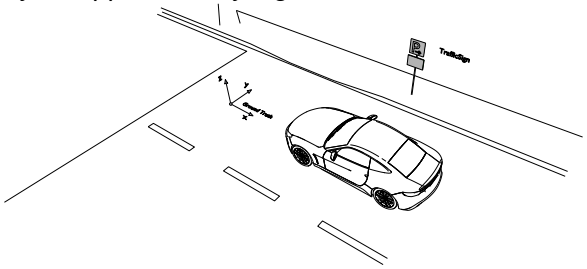


Figure 4.7 `TrafficSign`

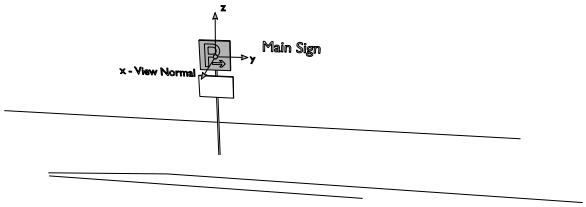


Figure 4.8 `MainSign`

TrafficSign	OSI	CityGML	IntIdx
id	value [uint64]	identifier_road_ObjectID	DA
main_sign	MainSign[struct]		
classification	Classification[struct]		
type	Type[enum]	opendrive_roadSignal_subtype, identifier_roadObjectName, opendrive_roadSignal_type	CM
vertically_mirrored	bool	opendrive_roadSignal_subtype	DA
is_out_of_order	bool	--	NP
assigned lane id	rep. value [uint64]	--	NA
direction_scope	DirectionScope[enum]	opendrive_roadSignal_subtype	CM
variability	Variability	--	NP
value	TrafficSignValue[struct]		
text	string	--	NA
value_unit	Unit[enum]	--	NA
value	double	opendrive_roadSignal_value	DA
base	BaseStationary[struct]		
orientation	Orientation3d[struct]	geometry_rotation_z, geometry_rotation_y, geometry_rotation_x	DA
dimension	Dimension3d[struct]	--	NA
position	Vector3d[struct]	lod1Geometry	DA
base_polygon	rep. Vector2d[struct]	--	NA
supplementary_sign	rep. SupplementarySign[struct]		
classification	Classification[struct]		
type	Type[enum]	opendrive_roadSignal_subtype, opendrive_roadSignal_type	CM
arrow	rep. Arrow[struct]	opendrive_roadSignal_subtype	CM
assigned_lane_id	rep. value[uint64]	--	NA
is_out_of_service	bool	--	NP
actor	rep. Actor[enum]	opendrive_roadSignal_subtype	CM
variability	Variability	--	NP
value	rep. TrafficSignValue[struct]		
text	string	--	NA
value_unit	Unit[enum]	--	NA
value	double	opendrive_roadSignal_value	DA
base	BaseStationary[struct]		
orientation	Orientation3d[struct]	geometry_rotation_z, geometry_rotation_y, geometry_rotation_x	DA
dimension	Dimension3d[struct]	--	NA
position	Vector3d[struct]	lod1Geometry	DA
base_polygon	rep. Vector2d[struct]	--	NA

IntIdx Interoperability Index (IntIdx): abbreviations can be found in Table 4.1
bold text denotes a nested type
rep. denotes a repeated type

Table 4.11 Traffic Sign interoperability

Some annotations:

- 3 CityGML Types have to be combined to find the right sign in OSI. The concepts used in CityGML are nearly the same as the ones used in the German StVo. Those concepts differ quite a bit from the ones used in OSI. To map all sign types a lot of special cases have to be considered.

- The meaning of `opendrive_roadSignal_subtype` heavily depends on `opendrive_roadSignal_type` and sometimes seems not very logical and a bit unclear.
- CityGML does not contain any logical connection between a `MainSign` and its Supplementary signs. The connection could be calculated by considering the relative position and orientation between `MainSigns` and `SupplementarySigns`. Probably the semantics of each sign should also be considered, as there might be more than one main sign on a pole.

4.6 Traffic Light

The traffic light object describes a single light bulb. It can have different colors and statuses that can change. In CityGML only a static object is described that does not contain any status that may change. In CityGML the traffic light is described as a whole sign, containing at least one light.

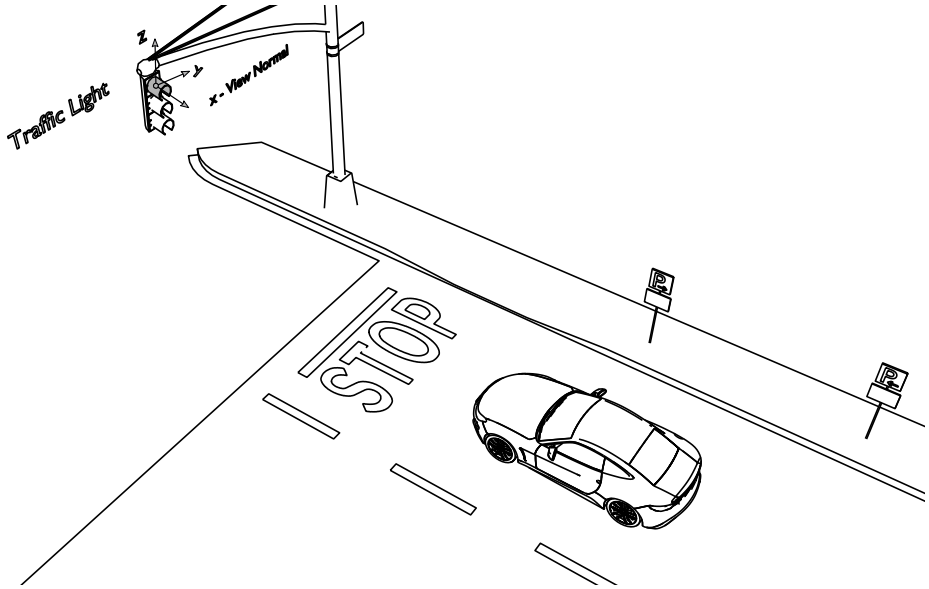


Figure 4.9 Illustration of the `TrafficLight`

TrafficLight	OSI type	CityGML	IntIdx
id	value [uint64]	identifier_road_ObjectID	DA
classification	Classification[struct]		
mode	Mode[enum]	--	NA
is_out_of_order	bool	--	NP
color	Color[enum]	opendrive_roadSignal_type	CM
icon	Icon[enum]	opendrive_roadSignal_subtype	CM
assigned_lane_id	rep. int	--	NA
counter	double	--	NP
base	BaseStationary[struct]		
orientation	Orientation3d[struct]	geometry_rotation_-z, geometry_rotation_y, geometry_rotation_x	DA
dimension	Dimension3d[struct]	--	NA[1]
position	Vector3d[struct]	lod1Geometry	CM
base_polygon	rep. Vector2d[struct]	--	NA[1]

IntIdx Interoperability Index (IntIdx): abbreviations can be found in Table 4.1
bold text denotes a nested type
rep. denotes a repeated type
1 since the geometry is a point object the dimension is zero

Table 4.13 Traffic Light

As already mentioned this object only describe the actual light source (bulb) and not the spatial object, like the pole the light is fixed to. CityGML contains a geometry object that is called `trafficLight`, which models the pole of a TrafficLight, but no relation is modeled in the document.
CityGML and OSI have different concepts when modeling a traffic light. To get the `color`, `icon` and `position` that is needed for the OSI object, one would have to do several steps:

- Recognize what Type it is by using `opendrive_roadSignal_type`.
- Depending on the Type and the `lod1Geometry` the `position` and `color` of the single light bulbs can be calculated.
- The icon can be calculated using `Type`, `position` and `opendrive_roadSignal_subtype`.

Using all the above information a non-complete OSI `TrafficLight` can be created.

4.7 Summary/Conclusion

CityGML / Opendrive has in many cases fundamentally different concepts compared to OSI. This seems natural keeping in mind that CityGML was extended by OpendDrive, while OSI was build to model traffic related sensor data. Another thing is that CityGML stores the entire data, while OSI allows exchanging information based on smaller messages between sensors and the simulation environment. The mapping of static attributes like IDs, names or geometries are straight forward and with some custom mapping possible in a lot of cases. The concept of dynamic attributes, like temperature or health status, does not (yet) exist in CityGML. Here future CityGML versions will solve this problem. It is nice to see that a lot of classes are found in both standards, so in many cases simple 1 to 1 mappings are possible. CityGML is a standard with many possibilities for custom mappings and extensions. Whereas OSI is much more restrictive, only allowing specific values defined by enumeration-types. In some cases the concepts of both standards lead to very elaborate custom mapping functions (e.g. traffic sign). Last but not least: both standards are in an open development process and in future versions a lot of the problems will be solved.

5 System Architecture/Software Design Patterns/Concepts

Our system consists of 3 major parts (see fig. 5.1). There is the database holding the CityGML data, the self-written program (`CityGML2OSI`) that converts the CityGML data into valid OSI data and the OSI-Visualizer that is responsible for displaying the data. The communication with the database is done using SQL, which has to be understood by our program. Afterwards OSI is utilized to communicate with the visualizer. In the following all parts are described in detail.

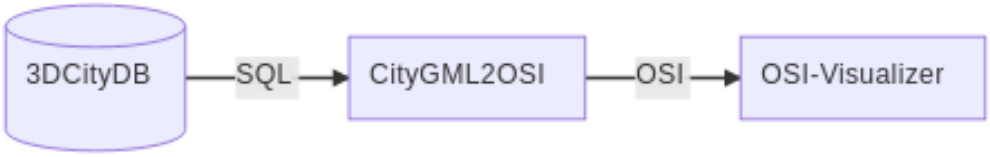


Figure 5.1 System Architecture Concept

5.1 Connection to the database

The previously mentioned 3D City Database (subsection 3.2.2) stores the provided data (section 3.3) in 66 tables that are modeled to represent the CityGML standard (subsection 3.1.1) in a database. To reduce the complexity and simplify queries, created with SQLAlchemy, materialized views get created.

5.1.1 View creation in the database as an interface

PostgreSQL, the relational database behind the 3DCityDB offers the concept of *Views*. They allow select-queries to be saved as a temporary table. The further development of the view is the materialized view, which not only stores the query itself, but also caches the data and therefore allows for faster access/execution times.
A problem that can be solved by this approach is posed by the `cityobject_genericattrib` table. Here generic attributes for a single `cityobject` are scattered over multiple rows. Each attribute gets its own row, where each column represents a different datatype and so most columns stay empty. Figure 5.2 details the afore mentioned problem.

id [PK]	attrname character varying (256)	datatype integer	strval character varying (4000)	intval integer	realval double precision	urival character	dateval timestamp	unit character	genattribset_codespace character varying (4000)	blobval bytea	cityobject_id integer
1	identifier_laneId	2	[null]	3	[null]	[null]	[null]	[null]	[null]	[null]	5
2	identifier_laneSectionId	2	[null]	0	[null]	[null]	[null]	[null]	[null]	[null]	5
3	identifier_roadId	1	1399000	[null]	[null]	[null]	[null]	[null]	[null]	[null]	5
4	identifier_modelName	1	892_SAVe_Ingolstadt	[null]	[null]	[null]	[null]	[null]	[null]	[null]	5
5	identifier_modelDate	1	13-06-17	[null]	[null]	[null]	[null]	[null]	[null]	[null]	5
6	identifier_modelVendor	1	3D Mapping Solutions	[null]	[null]	[null]	[null]	[null]	[null]	[null]	5
7	identifier_sourceFileName	1	2019-10-31_SAVe_Ingolstadt_P...	[null]	[null]	[null]	[null]	[null]	[null]	[null]	5
8	identifier_sourceFileExtension	1	xodr	[null]	[null]	[null]	[null]	[null]	[null]	[null]	5
9	identifier_sourceFileHashSha256	1	349efc0f107f6ed73d2039ac7a...	[null]	[null]	[null]	[null]	[null]	[null]	[null]	5
10	identifier_laneId	2	[null]	4	[null]	[null]	[null]	[null]	[null]	[null]	11

Figure 5.2 As defined in the 3DCityDB schema

This issue is resolved by creating the `genericattribs` materialized view inside the `osi` schema, as seen in Source Code 5.1. Here the lines get at first grouped by the `cityobject_id` and then the value, in this case `realval` gets aggregated with the `array_agg` function. Since the grouping naturally returns multiple rows, they have to be filtered to only return a single value. This is done by using the `FILTER` function and filtering by the attribute name in this case `geometry_rotation_x`. At the end each produced value gets a new alias that functions as the column name.

```
1  CREATE MATERIALIZED VIEW IF NOT EXISTS osi.genericattribs
2  TABLESPACE pg_default
3  AS
4   SELECT cityobject_genericattrib.cityobject_id,
5      (array_agg(cityobject_genericattrib.realval) FILTER (WHERE
       ↪  (((cityobject_genericattrib.attrname)::text = 'geometry_rotation_x'::text) AND
       ↪  (cityobject_genericattrib.realval IS NOT NULL))))[1] AS geometry_rotation_x,
6      ...
7     FROM cityobject_genericattrib
8    GROUP BY cityobject_genericattrib.cityobject_id
9  WITH DATA;
```

Source Code 5.1 Excerpt from `genericattribs` materialized view

Figure 5.2 shows the resulting table with the first 10 city objects and their corresponding attributes. This now allows queries to be less complex and provides a building block for the following views.

cityobject_id integer	identifier_laneId integer	identifier_laneSectionId integer	identifier_roadId character varying	identifier_modelName character varying	identifier_modelDate character varying	identifier_modelVendor character varying	identifier_sourceFileName character varying	identifier_sourceFileExtension character varying
1	2	0	1399000	892_SAVe_Ingolstadt	13-06-17	3D Mapping Solutions	2019-10-31_SAVe_Ingolstadt_Prio5	xodr
2	0	4	1399000	892_SAVe_Ingolstadt	13-06-17	3D Mapping Solutions	2019-10-31_SAVe_Ingolstadt_Prio5	xodr
3	0	10	1399000	892_SAVe_Ingolstadt	13-06-17	3D Mapping Solutions	2019-10-31_SAVe_Ingolstadt_Prio5	xodr
4	-2	0	1399000	892_SAVe_Ingolstadt	13-06-17	3D Mapping Solutions	2019-10-31_SAVe_Ingolstadt_Prio5	xodr
5	3	0	1399000	892_SAVe_Ingolstadt	13-06-17	3D Mapping Solutions	2019-10-31_SAVe_Ingolstadt_Prio5	xodr
6	0	16	1399000	892_SAVe_Ingolstadt	13-06-17	3D Mapping Solutions	2019-10-31_SAVe_Ingolstadt_Prio5	xodr
7	0	8	1399000	892_SAVe_Ingolstadt	13-06-17	3D Mapping Solutions	2019-10-31_SAVe_Ingolstadt_Prio5	xodr
8	0	0	1399000	892_SAVe_Ingolstadt	13-06-17	3D Mapping Solutions	2019-10-31_SAVe_Ingolstadt_Prio5	xodr
9	[null]	[null]	1399000	892_SAVe_Ingolstadt	13-06-17	3D Mapping Solutions	2019-10-31_SAVe_Ingolstadt_Prio5	xodr
10	1	0	1399000	892_SAVe_Ingolstadt	13-06-17	3D Mapping Solutions	2019-10-31_SAVe_Ingolstadt_Prio5	xodr

Figure 5.3 As defined in our `osi` schema

To further reduce the complexity, each OSI object gets its own materialized view. The following code shows such an implementation for the `StationaryObject`. Here the previously mentioned `genericattribs` view can be seen in action. Next to those generic attributes, the geometry is playing a major role. A lot of the columns are derived from the `surface_geometry` table, by the use of PostGIS functions. It needs to be noted here that it is not possible to derive the correct floor plans of buildings in a simple way, therefore they are approximated by the use of the `st_convexhull` function.

```
1  CREATE MATERIALIZED VIEW IF NOT EXISTS osi."StationaryObject"
2  TABLESPACE pg_default
3  AS
4    SELECT ga.cityobject_id,
5      ga."opendrive_roadObject_type" AS "Type",
6      st_force2d(st_convexhull(st_collect(sg.geometry))) AS basepolygon,
7      ga.geometry_rotation_x,
8      ga.geometry_rotation_y,
9      ga.geometry_rotation_z,
10     (st_xmax((st_collect(sg.geometry))::box3d) - st_xmin((st_collect(sg.geometry))::box3d)) AS width,
11     (st_ymax((st_collect(sg.geometry))::box3d) - st_ymin((st_collect(sg.geometry))::box3d)) AS length,
12     (st_zmax((st_collect(sg.geometry))::box3d) - st_zmin((st_collect(sg.geometry))::box3d)) AS height,
13     st_x(st_centroid(st_collect(sg.geometry))) AS centroid_x,
```

```
    st_y(st_centroid(st_collect(sg.geometry))) AS centroid_y,
    (((st_zmax((st_collect(sg.geometry))::box3d) - st_zmin((st_collect(sg.geometry))::box3d)) /
    ↪  (2)::double precision) + st_zmin((st_collect(sg.geometry))::box3d)) AS centroid_z,
    ga."identifier_modelName" AS model_reference
   FROM (osi."genericattribs" ga
     JOIN surface_geometry sg ON ((ga.cityobject_id = sg.cityobject_id)))
  WHERE ((ga."opendrive_roadObject_type" IS NOT NULL) AND ((ga."opendrive_roadObject_type")::text <>
  ↪  'NONE'::text))
  GROUP BY ga.cityobject_id, ga."opendrive_roadObject_type", ga.geometry_rotation_x,
  ↪  ga.geometry_rotation_y, ga.geometry_rotation_z, ga."identifier_modelName"
WITH DATA;
```

Source Code 5.2 Materialized View example based on the `StationaryObject`

5.1.2 Object Relational Mapper: SQLAlchemy

To connect the materialized views from the database with the following Python code as explained in section 5.2 a Object Relational Mapper (ORM) is used, in this case it is the `SQLAlchemy` package. `GeoAlchemy` further extends this package with spatial capabilities, such as PostGIS offers.
A ORM is most commonly based on either the *Facade* or the *Adapter* design pattern.
Listing 5.3 shows the `SQLAlchemy` ORM in use defining the `StationaryObject` class. It not only speeds up the development of Python classes with a connection to a database, but also offers the advantage of editor intelligence, such as auto-complete and linting. This boilerplate code can also be generated by tools such as `sqlacodegen`, which unfortunately did not work for materialized views.

```python
class StationaryObject(Base):
    __tablename__ = 'StationaryObject'
    __table_args__ = {'schema': 'osi'}

    cityobject_id = Column(Integer, primary_key=True)
    Type = Column(String)
    basepolygon = Column(Geometry(from_text='ST_GeomFromEWKT', name='geometry'))
    geometry_rotation_x = Column(Float(53))
    geometry_rotation_y = Column(Float(53))
    geometry_rotation_z = Column(Float(53))
    width = Column(Float(53))
    length = Column(Float(53))
    height = Column(Float(53))
    centroid_x = Column(Float(53))
    centroid_y = Column(Float(53))
    centroid_z = Column(Float(53))
    model_reference = Column(String)

    def geom(self):
        print(f"{list(to_shape(self.basepolygon).exterior.coords)}")
```

Source Code 5.3 Excerpt of `StationaryObject` ORM connection class from `osidb.py`

5.2 CityGML2OSI

Since OSI uses protocol buffer, the messages and objects must be constructed as protocol buffers as well. The concept how to convert the resulting object of the ORM into OSI `Ground Truth` message, in especially into the respective OSI class of interest, is explained in the following subsections. Therefore, Source Code 5.4, presenting an excerpt of the `CityGML2OSI - StationaryObject`, is extended step-wise, in order to demonstrate the data-flow. For the sake of simplicity is the number of attributes in contrast to real python implementation of `CityGML2OSI - StationaryObject` reduced (denoted by "..."), as the principle can be explained with these two attributes too. Source Code 5.4 lists only a standard constructor to store the values of interest as attributes.

```
1  import typing
2
3
4  class StationaryObject:
5      """
6      The base attributes of stationary object or entity
7      """
8      orientation: Orientation3d
9      base_polygon: typing.List[Vector2d]
10     ...
11
12     def __init__(self, orientation: Orientation3d = None, base_polygon: typing.List[Vector2d] = None)
       ↪  -> None:
13         if orientation is not None:
14             self.orientation = orientation
15
16         if base_polygon is not None:
17             self.base_polygon = base_polygon
18         ...
19
20         return None
```

Source Code 5.4 CityGML2OSI - Constructor of StationaryObject

5.2.1 from_sql

At first the ORM resulting object must be fed into the `CityGML2OSI - StationaryObject`. Therefore, the class is extended by a static method `from_sql()`, which directly consumes the ORM object and creates an object instance (see Source Code 5.5). A further advantage of this approach is, that individual attributes of different types, like `Orientation3d`, do not have to be parsed and created before each instantiation. This modelling approach is comparable to the factory-pattern, which is described by Gamma (1995), as the explicit constructor call is delegated to a factory-method, namely `from_sql()`, deciding which object to construct. This procedure can be seen in Source Code 5.5 between line 24 and 30, because different constructors are called based on the availability of the individual attributes.

```
1  import typing
2
3
4  class StationaryObject:
5      """
6      The base attributes of stationary object or entity
7      """
8      orientation: Orientation3d
9      base_polygon: typing.List[Vector2d]
10     ...
```

```
11
12      def __init__(self, orientation: Orientation3d = None, base_polygon: typing.List[Vector2d] = None)
        ↪  -> None:
13          if orientation is not None:
14              self.orientation = orientation
15
16          if base_polygon is not None:
17              self.base_polygon = base_polygon
18          ...
19
20          return None
21
22      @staticmethod
23      def from_sql(tsql: typing.Any) -> 'StationaryObject':
24          return StationaryObject(
25              orientation=Orientation3d(tsql.geom_rot_x, tsql.geom_rot_y, tsql.geom_rot_z)
26                  if hasattr(tsql, "geom_rot_x") else None,
27              base_polygon=list(map(
28                  lambda pt: Vector2d(x=pt[0], y=pt[1]),
29                  list(to_shape(tsql.basepolygon).exterior.coords))
30                  if hasattr(tsql, "basepolygon") else None
31              ...
32              )
```

Source Code 5.5 CityGML2OSI - StationaryObject construct via from_sql

5.2.2 write_pb

In the next step all the `CityGML2OSI (c2o)` objects are added as attributes into one `CityGML2OSI - Ground Truth` object. This object, being on top of the class hierarchy, is able to write the included information via the `write_pb()` into a protocol buffer defining the `OSI Ground Truth` message. In detail, calls the top-level `c2o.GroundTruth.write_pb(gt_pb: osi_groundtruth.GroundTruth)` method the same function in all elements or nodes, for instance for every `c2o.StationaryObject`, which again calls the `write_pb(pb)` and so on until the leaves are reached and the attributes are actually written into the passed OSI protocol buffer (pb). This complex design pattern, also known as composite pattern, which is defined by Gamma (1995), is required, because of the fact, that protocol-buffers can not be directly created with attributes, but must be instantiated and the attributes added afterwards manually.

Source Code 5.6 shows the principle of the composite pattern, as in each iteration in line 22 to 24 one empty `osi_vector2d` is added to `bs_pb` and filled with the call of `write_pb(osi_vector2d)`.

```
1   import typing
2
3
4   class StationaryObject:
5       """
6       The base attributes of stationary object or entity
7       """
8       orientation: Orientation3d
9       base_polygon: typing.List[Vector2d]
10      ...
11
12      def __init__(self, orientation: Orientation3d = None, base_polygon: typing.List[Vector2d] = None)
        ↪  -> None:
13          ...
14
15          return None
16
```

```
17      def write_pb(self, bs_pb: osi_common.StationaryObject) -> None:
18          if hasattr(self, "orientation"):
19              self.orientation.write_pb(bs_pb.orientation)
20
21          if hasattr(self, "base_polygon"):
22              for element in self.base_poylgon:
23                  osi_vector2d = bs_pb.base_polygon.add()
24                  element.write_pb(osi_vector2d)
25
26          return None
```

Source Code 5.6 CityGML2OSI - StationaryObject write to Protobuffer

6 Validation and Visualization

An important aspect of our project was the validation and even more the visualization of the results. The validation was done with the ASAM OSI-Validator. To visualize the data there were some tools to choose from. The first utilized tool was the ASAM OSI-Visualizer. But we also wanted to use a more sophisticated tool, so we had a look at Carla, but in the end we decided to use the PMSF OSI3Viewer. In the following all tools are described in detail.

6.1 ASAM OSI-Validator

ASAM e.V. the governing organization of the OSI standard also coordinates the development of the *OSI-Validator*. This tool allows next to the syntactic validation also the semantic verification with the help of rules. Those rules need to be generated with the `python ruels2yml.py` command or written manually into the `rules.yml` file.
Figure 6.1 shows the validation result of *TrafficLight*s. While some errors could be attributed to the issues mentioned in section 4.6, others resulted from missing attributes in provided data as well as an incomplete modeled `GroundTruth`.

```
Instantiate logger ...
Reading data ...
Retrieving messages in osi trace file until 26715 ...
1 messages has been discovered in 0.00021958351135253906 s
Collect validation rules ...

Caching ...
Importing messages from trace file ...
Caching done!

Errors (21)
  Ranges of timestamps  Message
----------------------  ------------------------------------------------------------------------------------
                     0  GroundTruth.host_vehicle_id.is_set(None) does not comply in GroundTruth
                     0  GroundTruth.country_code.is_set(None) does not comply in GroundTruth
                     0  GroundTruth.version.is_set(None) does not comply in GroundTruth
                     0  GroundTruth.timestamp.is_set(None) does not comply in GroundTruth
                     0  GroundTruth.stationary_object.is_set(None) does not comply in GroundTruth
                     0  GroundTruth.moving_object.is_set(None) does not comply in GroundTruth
                     0  GroundTruth.traffic_sign.is_set(None) does not comply in GroundTruth
                     0  BaseStationary.dimension.is_set(None) does not comply in GroundTruth.traffic_light.base
                     0  BaseStationary.position.is_set(None) does not comply in GroundTruth.traffic_light.base
                     0  BaseStationary.orientation.is_set(None) does not comply in GroundTruth.traffic_light.base
                     0  BaseStationary.base_polygon.is_set(None) does not comply in GroundTruth.traffic_light.base
                     0  TrafficLight.base.is_valid(None) does not comply in GroundTruth.traffic_light.base
                     0  TrafficLight.classification.is_set(None) does not comply in GroundTruth.traffic_light
                     0  GroundTruth.traffic_light.is_valid(None) does not comply in GroundTruth.traffic_light
                     0  GroundTruth.road_marking.is_set(None) does not comply in GroundTruth
                     0  GroundTruth.lane_boundary.is_set(None) does not comply in GroundTruth
                     0  GroundTruth.lane.is_set(None) does not comply in GroundTruth
                     0  GroundTruth.occupant.is_set(None) does not comply in GroundTruth
                     0  GroundTruth.environmental_conditions.is_set(None) does not comply in GroundTruth
                     0  GroundTruth.proj_string.is_set(None) does not comply in GroundTruth
                     0  GroundTruth.map_reference.is_set(None) does not comply in GroundTruth

Warnings (0)
Ranges of timestamps    Message
----------------------  ---------
```

Figure 6.1 ASAM OSI-Validator

6.2 ASAM OSI-Visualizer

Similar to the ASAM OSI-Validator, the *OSI-Visualizer*'s development is coordinated a governed by the ASAM e.V. organization. It supports GroundTruth, SensorView and SensorData messages and allows the visualization of two independent data channels using the OSI file or a stream.
For visualization as seen in Figure 6.2 the same file as in section 6.1 is used. The program was build to visualize the ground truth and what the vehicle is "seeing". Therefore, the User-Interface (UI) of the tool is split horizontally into two channels. The upper channel is the sending part of the tool. It reads displays and broadcasts the raw OSI files. In figure 6.2 one can see on the upper view the position of "Traffic Signs" with labels attached to them. To the left of the view there is a tree view of all objects. To the right of the view is panel with all settings. Here the OSI file can be selected and settings concerning the broadcasting of the data.
The lower part of the tool (see fig. 6.2) is the receiving channel. The view shows what the vehicle is seeing. On our case we did not specify any vehicle, so the view is empty. The settings to the right of the view define how the data is received.
The tool can be used to get a first impression of the created data. But it does not really fit our needs. The differentiation between sending and receiving channel makes things more complicated. Also, the limited visualization capabilities (only 2D) makes it hard to actually evaluate the quality of the files we have created.

Figure 6.2 ASAM OSI-Visualizer

6.3 PMSF IT-Consulting – FMI Bench

The PMSF OSI3Viewer is a tool developed by the company PMSF IT-Consulting, it is part of the FMI Bench developed by the same company. The tool is an alternative to the previously introduced OSI Visualizer, featuring 3D rendering (see fig. 6.3). It is written in the python language and therefore gave us the ability to directly implement an interface to our CityGML2OSI software. The 3D capabilities of the tool allow for a much better inspection and validation of the data. The camera automatically follows the vehicle and can be freely rotated and zoomed, to give the user a good idea of the data. The visualization of the vehicle also features indications of the field of view of the sensors.

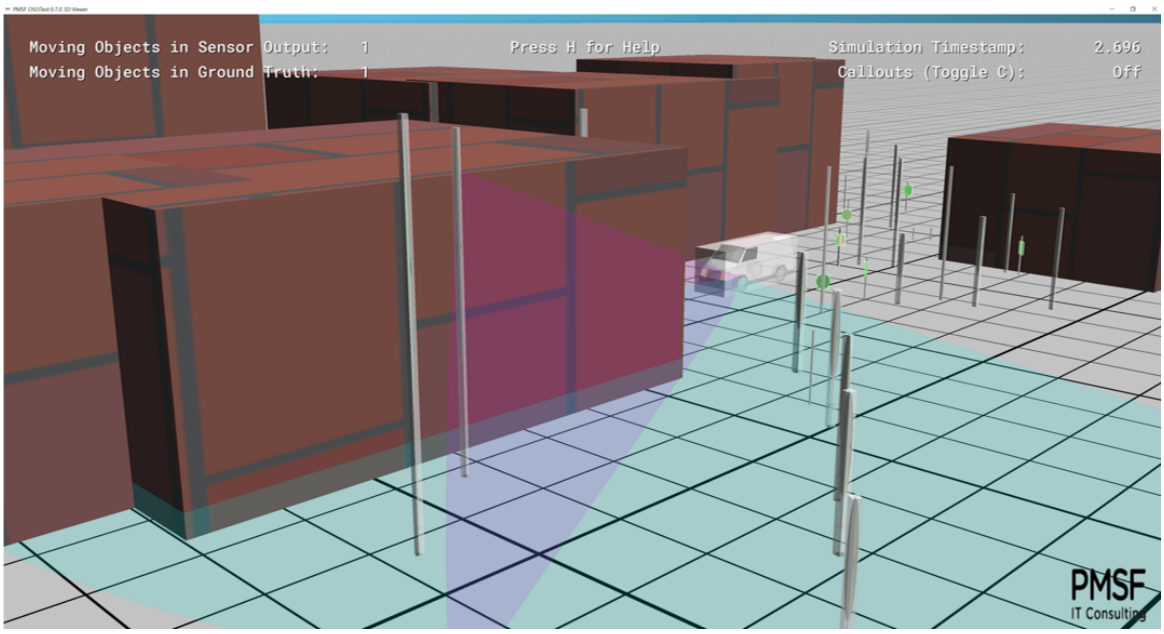


Figure 6.3 PMSF IT-Consulting

6.3.1 Known issues and workarounds

The product is still work in progress and therefore has some limitations and problems. During our work with the tool several observations where made:

- **"*Wiggling*" - Too large coordinates**
  All positions in the data are given in UTM country coordinates that are quite long (6 digits in front of the decimal separator). As a result we observed "*wiggling*" of the objects, they changed their position in every new frame, when using offset coordinates (4 digits in front of the decimal separator) this strange behavior was gone. This could be caused by the issues and limitations of floating point operations in Python (Python Software Foundation, 2021).

- **Wrongly defined origin**
  Another issue is the different approach of visualizing a model as OSI defines the model with respect to the 3D center of mass or `position` with $z = z_{min} + \frac{z_{max} - z_{min}}{2}$, whereas the FMI Bench seems to utilize the horizontal center of mass with $z = z_{min}$ and places the model on top of it, which might lead to some height problems in the final visualization.

- **Limited number of implemented types** (traffic signs, walls as buildings)
  There are still a lot of objects that are not supported by the tool yet. It automatically renders traffic signs with the right dimensions and textures (an image of the actual sign). Until now, only a subset of Straßenverkehrsordnung (StVo) compliant signs are supported in that way. Also, buildings are not implemented as objects. As a workaround we used the already supported walls to display our buildings, resulting in those strange textures (see Figure 6.3) and wrong orientations.

- **Problematic visualization of lanes**
  Another issue comes with the use of lanes and lane-boundaries, as they are only supported in a simplified and limited way, without a lot of core functionalities. As the visualizer derives the geometry of the boundaries, but does not seem to consider the center-line of the lane itself, which leads to problems with very short lanes that are defined by few points.

7 Conclusion

The main task of the project was importing CityGML Data via OSI into a driving simulator.
In order to achieve this goal, the task was split into the following 5 parts:

- **Analysis of provided CityGML Data**: The software FME was used to visually inspect the provided data.
- **Interoperability Analysis**: A very detailed inspection of 6 different objects was performed. The results of the comparison of OSI objects to CityGML objects were written down into tables and discussed.
- **Development Environment**: A sophisticated development environment was set up. Allowing seamless development and testing with the whole team, software packages like Docker and Git were used.
- **Developing a CityGML to OSI converter**: A python program was written. This program provides an interface that connects to the Database holding the CityGML data. After the conversion, another self written interface is used to stream OSI data to a visualization / simulation software.
- **Visualization**: Two different visualization programs were tested. The ASAM OSI-Visualizer could display our data by reading an OSI file we created. A special interface was implemented to stream OSI-Data directly to the PMSF OSI3-Viewer.

This gave us a good opportunity to get to know different data types in detail and to understand the different needs of both acting parties. While CityGML tries to store and exchange structured information of virtual 3D city models on the one hand. OSI on the other hand is a generic interface between a sensor and the ground truth (parts of the virtual city model). The in-depth interoperability analysis of the standards showed us how complex they are. This also let us to invest more time than we initially expected. The limited amount of validation and visualization options turned out to be a real problem. During most of the development we had no option of verifying our results visually. Only the software from PMSF we acquired during the end of the project allowed for visual verification of our work.

The field of driving simulations is still a very young one, that is why no real standards have emerged yet, CityGML, OSI and OpenDrive are still in their infancy. Even during the time we worked on this project, the new CityGML3.0 standard was released that contained a lot of improvements for modeling dynamic objects. Sadly in the scope of the project it was not possible to use the new standard. During 2021 the OSI4 standard will be released. Next to further extensions and major changes for handling static objects, the consistency with OpenDrive will be improved a lot.

We will publish our work on GitHub (`https://github.com/tum-gis/citygml2-to-osi3`) so that others can use the code and continue the development. In case you want to continue this project, there are multiple starting points:

- Extend the library by further mappings and new OSI messages to build a complete plugin which might be used in combination with an OSI-viewer.
- Study the upcoming standards, which might increase the interoperability.
- Return geometries from the database instead of the individual coordinate components to simplify the coordinate issue.

Glossary

3DCityDB Geo-database to store, represent, and manage virtual 3D city models on top of a standard spatial relational database.

ASAM e.V. Association for Standardization of Automation and Measuring Systems

BMW Bayerische Motoren Werke

CityGML Open data model and XML-based format for the storage and exchange of virtual 3D city models.

FME Data Inspector

GIS Geographic Information System

GML Geography Markup Language

LOD Level of Detail

OpenDRIVE

ORM Object Relational Mapper

OSI Open Simulation Interface

OSM Open Street Map

PGAdmin

PostGIS GIS enhancement of the PostgreSQL database

r:trån a road space model transformer for OpenDRIVE to CityGML

StVo Straßenverkehrsordnung

TUM Technical University Munich

UI User-Interface

UTM Universal Transverse Mercator coordinate system

XML Extensible Markup Language

List of Tables

List of Figures

Listings

Bibliography

ASAM e.V. (2020). *ASAN OpenDRIVE: Open Dynamic Road Information for Vehicle Environment*.

CityGML 2.0: OGC City Geography Markup Language (CityGML) Encoding Standard (n.d.). URL: https://www.ogc.org/standards/citygml (visited on).

Docker Inc. (2021). *What is a Container? A standardized unit of software*. URL: https://www.docker.com/resources/what-container (visited on 01/23/2021).

Gamma, E. (1995). *Design patterns: Elements of reusable object-oriented software / Erich Gamma ... [et al.]* Addison-Wesley professional computing series. Reading, Mass. and Wokingham: Addison-Wesley. ISBN: 0201633612.

Google Developers (2008). *Protocol Buffers*.

Hanke, T., N. Hirsenkorn, C. van-Driesten, P. Garcia-Ramos, M. Schiementz, S. Schneider, and E. Biebl (2017). *A generic interface for the environment perception of automated driving functions in virtual scenarios*. URL: https://www.asam.net/standards/detail/osi/ (visited on 11/16/2020).

Kalra (2016). *Driving to Safety: How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability?* RAND Corporation. ISBN: 9780833095299. DOI: 10.7249/RR1478.

pgAdmin Development Team (2021). *Development*. URL: https://www.pgadmin.org/development/ (visited on 01/23/2021).

Python Software Foundation (2021). *15. Floating Point Arithmetic: Issues and Limitations*. URL: https://docs.python.org/3/tutorial/floatingpoint.html (visited on 02/25/2021).

Schwab, B., C. Beil, and T. H. Kolbe (2020). "Spatio-Semantic Road Space Modeling for Vehicle–Pedestrian Simulation to Test Automated Driving Systems". In: *Sustainability* 12.9, p. 3799. DOI: 10.3390/su12093799.

Thomsen, T. (2018). *OpenX-Standards at ASAM*. URL: https://www.asam.net/index.php?eID=dumpFile&t=f&f=2049&token=86c6617b9c8b8517e115f889451ad4d60b43d650.

Yao, Z., F. Kunde, S. H. Nguyen, C. Nagel, and T. H. Kolbe (2021). "3D City Database for CityGML: Release 4.2.3". In: URL: https://3dcitydb-docs.readthedocs.io/_/downloads/en/latest/pdf/ (visited on 02/12/2021).